Bringing declarative programming to Life

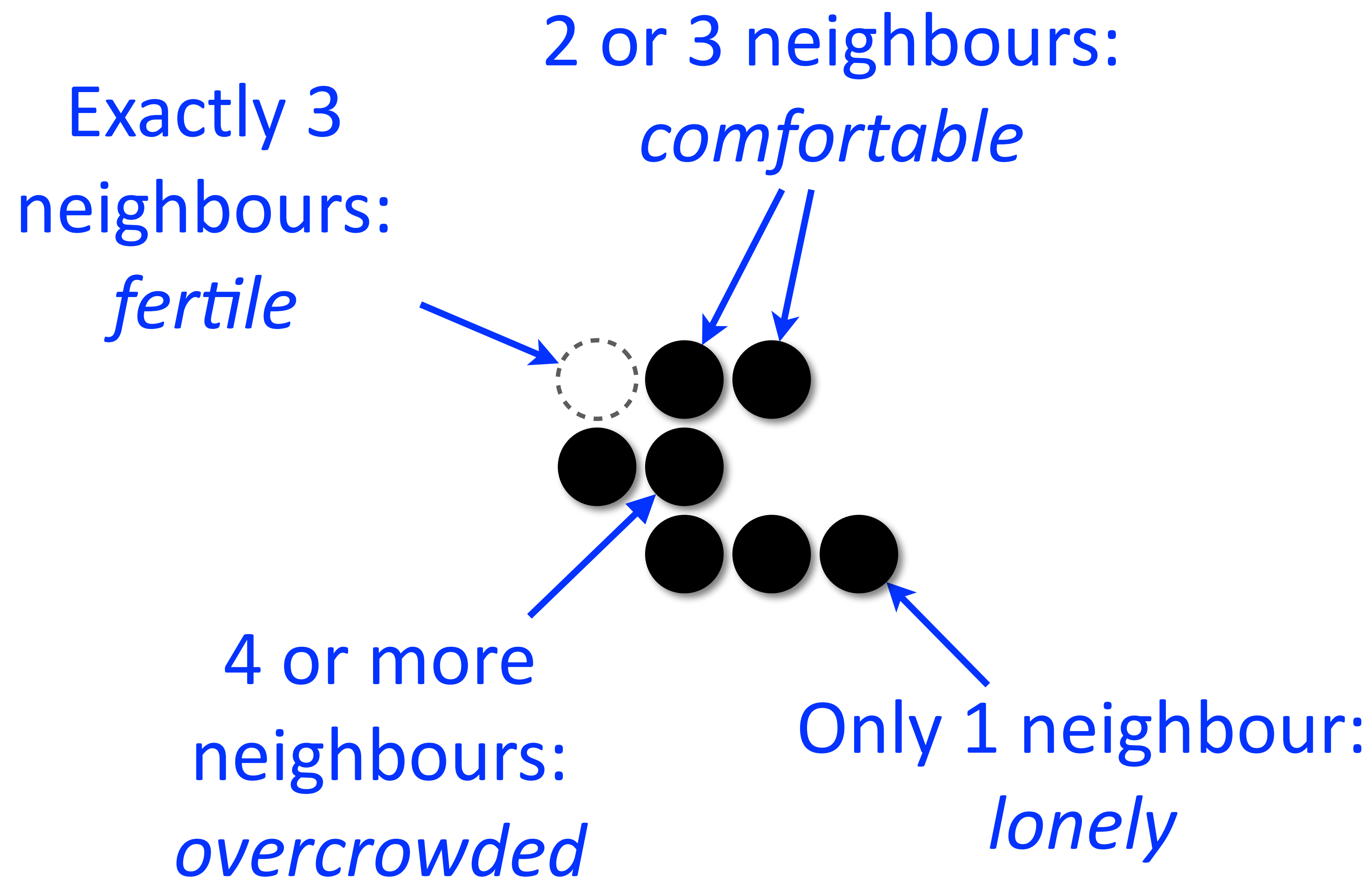
Michael Spivey

League for Side-Effect Freedom

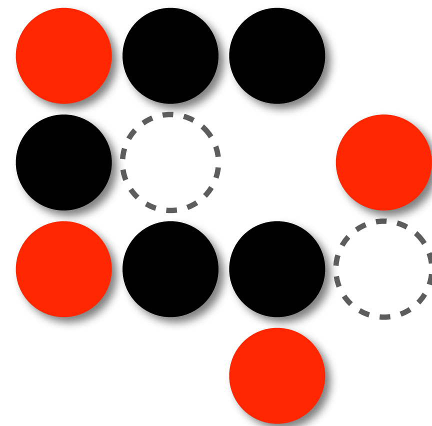
Department of
COMPUTER
SCIENCE

Copyright © 2016 J. M. Spivey

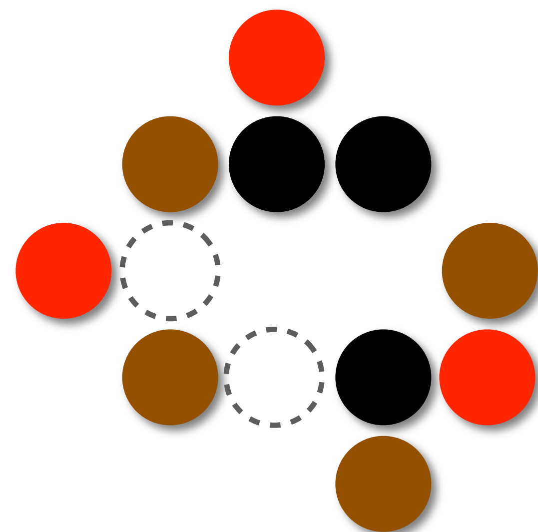
The game of Life



The next generation …



… and another one



Life in the Wolfram language

"In three lines of code"

```
gameOfLife = {224, {2, {{2, 2, 2}, {2, 1, 2},
    {2, 2, 2}}}, {1, 1}};

board = RandomInteger[1, {50, 50}];

Dynamic[ArrayPlot[board = Last[CellularAutomaton
    [gameOfLife, board, {{0, 1}}]]]]
```

Let's make our own!

One step at a time:

```
for t in range(0, 1000):
   # Compute the next generation
```

Let's make our own!

Update each cell:

```
for t in range(0, 1000):
    for x in range(0, N):
        for y in range(0, N):
            # Count the live neighbours
            # Update cell[x][y]
```

Let’s make our own!

Visiting the neighbours:

```
for t in range(0, 1000):
   for x in range(0, N):
      for y in range(0, N):
         score = 0
         for i in [-1, 0, 1]:
            for j in [-1, 0, 1]:
               if i <> 0 or j <> 0:
                  score += cell[x+i][y+j]
         score += cell[x][y]/2.0
         ...
```

Let's make our own!

Updating the cell:

```
for t in range(0, 1000):
    for x in range(0, N):
        for y in range(0, N):
            # Compute the score
            ...
            cell[x][y] =
                (score > 2 and score < 4)
```

A horrendous bug!

Try again!

The rules:

- Don’t write functions longer than three lines.
- Don’t use assignable variables.
- But do use predefined functions if they are *generally useful*.

I will use the language of GeomLab, but we could use Haskell, or at a pinch Python.

The master plan

Define a function *next*(*state*) that takes one state of the world and returns the next one.

Begin with an initial state *init*.

Then the Life history is

  [*init*, *next*(*init*), *next*(*next*(*init*)), ...]

going on as long as we want.

Using *repeat*

Suppose $repeat(n, f, x) = [x, f(x), f(f(x)), \ldots, f^n(x)]$:

> $repeat(10, (+2), 3) =$
> $[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]$.

Then the Life history is (say)

> $repeat(200, next, init)$.

We'll represent states by images, with pixels made from colours.

Working with colours

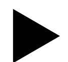
GeomLab has a few predefined colours:

black, *white*, *red.*

But other colours can be made as

$rgb(r, g, b)$,

where $0 \leq r \leq 1$, $0 \leq g \leq 1$, $0 \leq b \leq 1$.



Working with images

An image *img* is a rectangular array of pixels.

So we can ask

width(*img*), *height*(*img*),

and if [*x*, *y*] are coordinates, we can ask

pixel(*img*, [*x*, *y*])

for the colour at that point.

Making new images

Suppose f is a function such that $f([x, y])$ is a colour:

then $image(w, h, f)$ is a $w \times h$ image img defined by

$$pixel(img, [x, y]) = f([x, y]).$$

This is a completely new image, not an image got by modifying an old one.

Is that going to be too inefficient?
Let's try it and see!

World states as images

The initial state has black pixels in carefully chosen places.

> **define** *init* =
>   **let** *c* = [[6,2], [7,3], ...] **in**
>   *image*(*N*, *N*, **function** (*p*)
>   **if** *member*(*p*, *c*) **then** *black* **else** *white*)

Making the state visible

Replace each pixel with a coloured blob:

> **define** *blobify*(*img*) =
> *above*([*beside*([*pixel*(*img*, [x,y])
> | *x* ← [0..*width*(*img*)-1]])
> | y ← *reverse*([0..*height*(*img*)-1])])

This arranges the blobs in a rectangular array to match the image.

The next-state function

To get the next state, make a *completely new* image showing the cells that are *viable* in this one.

define *next*(*state*) =
image(*N*, *N*, **function** (*p*)
if *viable*(*p*, *state*) **then** *black* **else** *white*)

Which cells are viable?

As before, we must count the neighbours

```
define viable(p, state) =
    let val(q) = if pixel(state, q) <> white then 1 else 0 in
    let s = sum([ val(q) | q ← region(p) ]) + val(p)/2 in
    s > 2 and s < 4

define region([x, y]) =
    [ [u, v] | u ← [x-1 .. x+1], v ← [y-1 .. y+1]
                                    when u <> x or v <> y ]
```

(This time we can describe the neighbours separately.) ▶

Interactive images

To show the evolution of Life, we must be able to make a picture that depends on time.

slide(**function** (*t*) *rgb*(1-*t*, 0, *t*))

For an animation that has a list of frames:

define *animate*(*frames*) =
  let *n* = *length*(*frames*) **in**
  slide(**function** (*t*) *nth*(*int*((*n*-0.001)***t*, *frames*));

(The 0.001 ensures we don't fall off the end of the film.)

Putting it all together

Make a list of states, turn each one into a picture, then show the pictures as a slider-controlled animation.

> **let** *life* = *map*(*blobify*, *repeat*(200, *next*, *init*)) **in**
> *animate*(*life*)

A program built from independent, reusable components – in one line of code!

▶

Functional programming

- No ‘programming’ variables – use mathematical variables instead.
- No loop commands – use recursion instead.

Two benefits:

- Easier to make programs from independent pieces.
- Easier to reason about them mathematically.

Abstract data types

Sets of values (like our images) with operations defined on them.

- The operations obey rules that can be described algebraically. For example,

$$pixel(image(w, h, f), p) = f(p)$$

- The computer representation of values is hidden.

The GeomLab site

`http://www.cs.ox.ac.uk/geomlab`

The software:

- runs from the web page (if you're lucky).
- Java-based – no installation required.
- works perfectly on Raspberry Pi.

Teaching materials:

- full set of worksheets for a 1-2 day activity.